**Structures and Interpretation of Computer Program**

**Exercise Chapter 2.4 Name:** Wan Huzaifah bin Wan Azhar

**Exercise 2.4.3 Data Directed Additivity**



* We can’t assimilate number? and same-variable? because there is no tag to identify that the data should do what.
* For example, assume number? is input into the table. What tagged data should the table retrieve so that it returns the procedure number?
* Same things for same-variable? The operator can be ‘a ‘b ‘x ‘y. Not having one type to identify what procedure to return will cause issue.
* The same thing cannot be said for else expression. The expression can return the correct derivative procedure by identifying the operator of the expression.
  + + will returns (make-sum (deriv (addend exp) var) (deriv (augend exp) var)))



(define (deriv exp var)

(cond ((number? exp) 0)

((variable? exp) (if (same-variable? exp var) 1 0))

(else ((get 'deriv (operator exp)) (operands exp) var))))

(define (install-deriv-sum-pkg)

(define (addend s) (cadr s))

(define (augend s) (caddr s))

(define (make-sum a1 a2) (list '+ a1 a2))

(define (deriv exp)

(make-sum (deriv (addend exp) var)

(deriv (augend exp) var)))

;; Interface to rest of system

(put 'addend '+ addend)

(put 'augend '+ augend)

(put 'make-sum '+ make-sum)

(put 'deriv '+ deriv))

(define (install-deriv-mult-pkg)

(define (multiplier e) (cadr e))

(define (multiplicand e) (caddr e))

(define (make-product a1 a2) (list '\* a1 a2))

(define (deriv exp)

(make-sum

(make-product (multiplier exp)

(deriv (multiplicand exp) var))

(make-product (deriv (multiplier exp) var)

(multiplicand exp))))

;; Interface to rest of system

(put 'multiplier '\* multiplier)

(put 'multiplicand '\* multiplicand)

(put 'make-product '\* make-product)

(put 'deriv '\* deriv))

* I could have implements multiplicand that can handle more than two operands, but for this assignment, I think it is enough.
* Also, tag is not necessary here because the expression’s operator + and \* is already a type in itself.



(define (install-deriv-expt-pkg)

(define (exponentiation? exp)

(and (pair? x) (eq? (car x) '\*\*)))

(define (base s) (cadr s))

(define (exponent s) (caddr s))

(define (make-exponentiation base expnt)

(cond ((=number? expnt 0) 1)

((=number? expnt 1) base)

(else (list '\*\* base expnt))))

(define (deriv exp)

(make-product

(make-product (exponent exp)

(make-exponentiation (base exp)

(make-sum (exponent exp) -1)))

(deriv (base exp) var)))

;; Interface to rest of system

(put 'exponentiation? '\*\* exponentiation?)

(put 'make-exponentiation '\*\* make-exponentiation)

(put 'base '\*\* base)

(put 'exponent '\*\* exponent)

(put 'deriv '\*\* deriv))

(define (make-exponentiation base expt)

((get 'make-exponentiation '\*\*) base expt))

* Attaching tag is not necessary here because make-exponentiation will tag it if the evaluated expression required it. In essence, not every make-exponentiation will become exponent expression.



* All put argument in every packaged needs to be reordered.
* It is minimal changes to one package. However, keep in mind that all packages need to be rewrite. And rewritings sometimes leads to error.



(define (get-record employee file) ;return employee-record

((get 'get-record (which-division file)) employee file))

* Individual division file should be their own ‘get-record‘ procedure which can retrieve record of given employee.
* Each division should tag their division name in the file, so that the procedure which-division can find out what division is the file from and get the correct procedure to find record on their division.



(define (get-salary employee-record)

((get 'get-salary (which-division-record employee-record)) employee-record))

* Again, in each employee record, each division should tag their division name.
* Each division should also provide ‘get-salary’ procedure that returns employee salary given their employee record.



(define (find-employee-record employee list-division-file)

(if (null? (get-record employee (car list-division-file)))

(find-employee-record employee (cdr list-division-file))

(get-record employee (car list-division-file))))

* Every ‘get-record’ procedure on every division should return nothing if no employee is found from their file.



* Insatiable should create another tag type for the subcompany as the subcompany will have their own division and ways to get record. This will differentiate procedures of main-company and subcompany.
* Insatiable should also provide installation package for the subcompany to install their procedures.



(define (make-from-mag-ang r a)

(define (dispatch op)

(cond ((eq? op 'real-part)

(\* r (cos a)))

((eq? op 'img-part)

(\* r (sin a)))

((eq? op 'magnitude) r)

((eq? op 'angle) a)

(else

(error "Unknown op -- MAKE-FROM-MAG-ANG" op))))

dispatch)



* Explicit dispatch: To add new type or operation, each of new data type must be tagged appropriately and the dispatch procedure must be modified to support new operation on the new type.
* Data-directed: To add new operation, the procedure needs to be installed into the proper database so that all user can use it. The installed procedure must also provide a way to tag the data and all of the data must be tagged first before the procedure can be used.
* Message-passing: To add new operation, the dispatch procedure must be updated. The data also needs to be tagged.



* Data-directed style is the best way for organization that most often add new types.
* This is because in data-directed, new procedures and type are installed into the database.
* Therefore, any new changes to the database will not affect existing procedures.
* Any problem with new type can also be uninstalled without affecting other types.



* Data-directed style is still the best way for organization that most often add new operation.
* The reasoning is the same as b.
* Failure in new operation can be reversed without any consequences.